DOCUMENT RESUME

ED 057 582 EM 009 422

AUTHOR	Weiner, Walter B.; And Others
TITLE	The LOGO Processor: A Guide for System Programmers.
INSTITUTION	Bolt, Beranek and Newman, Inc., Cambridge, Mass.
SPONS AGENCY	National Science Foundation, Washington, D.C.
REPORT NO	B-2165
PUB DATE	30 Jun 71
NOTE	61p.; Programming-Languages as a Conceptual Framework for Teaching Mathematics; See also EM 009 419, EM 009 420, EM 009 421
EDRS PRICE	MF-$0.65 HC-$3.29
DESCRIPTORS	*Computer Assisted Instruction; *Computer Programs; *Mathematics Instruction; Program Descriptions; Programming Languages
IDENTIFIERS	LOGO Program

ABSTRACT

A detailed specification of the LOGO programing system is given. The level of description is intended to enable system programers to design LOGO processors of their own. The discussion of storage allocation and garbage collection algorithms is virtually complete. An annotated LOGO system listing for the PDP-10 computer system may be obtained on request. For further information about the LOGO program of computer-assisted instruction in mathematics see volumes I, II, III of the report (EM 009 419, EM 009 420, and EM 009 421). (JY)

BOLT BERANEK AND NEWMAN INC

CONSULTING • DEVELOPMENT • RESEARCH

Report No. 2165 Volume 4

PROGRAMMING-LANGUAGES AS A CONCEPTUAL FRAMEWORK FOR TEACHING MATHEMATICS

The LOGO Processor

A Guide for System Programmers

Submitted to:

National Science Foundation
Office of Computing Activities
1800 G Street, N.W.
Washington, D. C. 20550

Contract NSF-C 615

30 June 1971

Volume 4

The LOGO Processor

A Guide for System Programmers

The LOGO Project

NSF-C 615

Walter B. Weiner
Charles R. Morgan
Wallace Feurzeig

Bolt Beranek and Newman Inc.
50 Moulton Street
Cambridge, Mass. 02138

FOREWORD

This volume gives a detailed specification of the LOGO programming system. The level of description is intended to enable system programmers to design LOGO processors of their own, and to carry implementation through to within the innermost levels of "picky" details. Thus, while the discussion of storage allocation and garbage collection algorithms is virtually complete, the reader will not find the answers to myriads of questions like: what happens if one erases a nonexistent procedure? Questions at this level of detail are best answered by looking at the system listings directly.

The most complete descriptions, the annotated LOGO system listings of existing versions of LOGO, may be obtained on request. We have We have already provided program listings (and, in fact, actual programs) to a number of university research centers with PDP-10 computer systems.

The BBN PDP-10 LOGO system was implemented by Walter B. Weiner with the assistance of Paul Wexelblat and Charles R. Morgan. Earlier versions were implemented on the PDP-1 in LISP by Daniel Bobrow and in assembly language by Richard Grant and Charles R. Morgan, assisted by Paul Wexelblat. This document was written by Walter B. Weiner and Charles R. Morgan. Wallace Feurzeig contributed extensively to the organization and exposition.

The LOGO Processor
A Guide for System Programmers

CONTENTS

0. Introduction

Since its introduction in 1967, interest in the LOGO programming language has grown steadily and rapidly. Particular interest has been shown by mathematicians, computer scientists, and educators. Individuals representing several institutions have asked us how they could get LOGO processors on their own computer systems.

We seek, through this document, to help make LOGO generally available by elaborating a virtually complete design for a LOGO processor. We have tried to describe the processor algorithms plainly and clearly so as to make the underlying operation of LOGO accessible to researchers and teachers, as well as system programmers. At the same time, we have provided more-detailed information for use by system programmers who will be implementing LOGO processors.

The structure of LOGO programs and data imposes special requirements on the design of a reasonably efficient processor. An essential aspect of the LOGO language is the connective structure of user-defined procedures. Procedures can be recursively embedded and chained -- in principle to arbitrary depth. Thus, LOGO programs can generate a long string of procedure links (a large control stack). Moreover, programs can generate a long list of strings of varied length. Care is therefore required in the design to provide efficient use of both time and space. The discussion and examples throughout the text illustrate the rationale for specific choices made to satisfy these general requirements.

The main body of the document describes the algorithms for parsing and execution of LOGO instructions, and for storage allocation and garbage collection of LOGO programs and data. (Flow diagrams for the main algorithms are appended.) The discussion leads to the remarkable result that a single LOGO processor design is virtually universal and near-optimal.

The appendices include lists of LOGO commands, operations, names, and abbreviations for a complete and a minimal LOGO.

1. Processing an Input Line

All data in LOGO are represented as strings. This includes literals, names, abbreviations, comments, and procedure names. It also includes outputs of operations and inputs to operations, commands, and procedures. Input lines are also strings. An input line is a string of characters terminated by a carriage return or any other line terminator. This string of characters comes from either the terminal or a file. When a line is complete, all redundant spaces are removed. The line is then parsed to identify and separate the elements.

The elements include literals, names, comments, names of commands and operations, names of procedures defined by the user, and noise words such as OF, AND, AS, (,), used to clarify expressions. The following example shows how an input line is parsed. The type of each element is shown in parentheses.

```
PRINT FOO OF BUTFIRST OF "THE QUICK BROWN FOX"
```

	Element	Type
(1)	PRINT	(COMMAND)
(2)	FOO	(PROCEDURE)
(3)	OF	(NOISE WORD)
(4)	BUTFIRST	(OPERATION)
(5)	OF	(NOISE WORD)
(6)	"THE QUICK BROWN FOX"	(LITERAL)

After the input line is parsed, it is executed as follows.

(1) Fetch the first element PRINT: it needs one input.

(2) Fetch the next element FOO. This is a procedure previously defined by the user as follows.

```
←TO FOO /ANYTHING/
>1 PRINT /ANYTHING/
>2 OUTPUT BUTLAST OF /ANYTHING/
>END
```

FOO needs one input.

(3) Fetch the next element OF, noise word (i.e., has no effect). It is valid here.

(4) Fetch the next element BUTFIRST: it needs one input.

(5) Fetch OF, noise word. It is valid here.

(6) Fetch the literal "THE QUICK BROWN FOX". This is the input for BUTFIRST.

(7) Invoke BUTFIRST. It outputs "QUICK BROWN FOX". This is the input for FOO.

(8) Invoke FOO. Make its input, /ANYTHING/, the literal "QUICK BROWN FOX".

(9) Fetch the first instruction line of FOO (PRINT /ANYTHING/).

(10) Fetch the first element of this line, PRINT: it needs one input.

(11) Fetch the next element of the line, /ANYTHING/. This is a name. Its value, "QUICK BROWN FOX", is the input to PRINT.

(12) Invoke PRINT. PRINT prints out its input. It has no output.

(13) Fetch the next instruction line of FOO (OUTPUT BUTLAST OF /ANYTHING/).

(14) Fetch the first element of this line, OUTPUT: it needs one input.

(15) Fetch the next element of the line, BUTLAST: it needs one input.

(16) Fetch the next element of the line, OF. This is a noise word. It is valid here.

(17) Fetch the next element of the line, /ANYTHING/. This is a name. Its value, "QUICK BROWN FOX", is the input to BUTLAST.

(18) Invoke BUTLAST. It outputs "QUICK BROWN".

(19) Invoke OUTPUT. It outputs its input, "QUICK BROWN", and terminates execution of the procedure FOO.

(20) Invoke PRINT. PRINT prints out its input, the output of FOO, "QUICK BROWN". It has no output.

(21) Done. (The execution phase is complete.)

As shown above, LOGO processes an input line element by element, fetching the elements it needs to invoke all the commands, operations, and procedures encountered. If the first element of the

input line is a number (that is, an unsigned integer), the line is not executed immediately but is stored with the procedure currently being defined. In either case, LOGO continues by processing the next input line.

2. Parsing a Line

The first stage in processing an input line consists in parsing the text string into a list of its elements. These elements include literals, names, comments, noise words, and names of commands, operations, and procedures. Literals are enclosed in quotes; names are enclosed in slashes; comments are enclosed in semicolons; noise words and names of commands, operations, and procedures are not enclosed in special marks. An element is built up by scanning the successive characters of the text string and accumulating them by using the following logic.

Look at the first character of the string to see if it is a space, ", /, ;, (,), or end of line character (EOL). If it is a space, ignore it. If it is a " or /, it denotes the beginning of a literal or name. In either of these cases, scan successive characters and accumulate them to form an element, stopping when the next " or / is seen, and ignoring certain spaces encountered along the way (those immediately after the beginning " or /, those immediately before the terminal " or /, and all those except the first space whenever several occur in succession). If an EOL character is encountered before a subsequent " or /, exit and generate an error message. After the element is completed, create an associated list element containing the type (literal or name) of the element just formed and a string pointer to it.

If the first character encountered in forming the next element is a ;, this denotes the beginning of a comment. Once again, build up the element string from successive characters just as with literals or names. In this case, however, there are two legal terminators -- an EOL as well as a closing ;. The list element associated with the string identifies the string as a comment and points to its location in storage.

If the first character encountered in forming the next element is a (or a), it denotes the beginning or end of a presumed expression. In either case, generate a list element identifying the specific character, i.e., the left or right parenthesis. This element subsequently will direct the EXECUTE subroutine to process the subexpression appropriately.

If the first character encountered in forming the next element is an EOL, and there was a line number associated with the text string, store the line in the procedure currently being defined or edited. (If there is no such procedure, type out an error comment.) Then, in any case, exit from the parsing procedure.

If the first character encountered in forming the next element is not one of these special delimiters, this means we are building up either the name of a command, operation, or procedure or a line number or a number literal. We proceed to build up a string element from successive characters until we encounter any one of the following delimiters -- space, (,), ", /, ;, EOL. These delimiters cannot be inside a number or a command, operation, or procedure name. (As in the case of literals, names, and comments, these built-up strings are stored without preceding or terminating delimiters.)

After the string is built up, we determine whether or not it is a number. A number consists entirely of digits except possibly for a leading + or -. If it is a number without a leading sign and it is the first element on the line, we indicate that what we are building up will be a stored line of a procedure definition. If it is not the first element of the line, it is a number literal and we handle it in the same way as a quoted literal.

If the string is not a number string, we check to see whether it is an abbreviation. To do this, we look for it in the user's abbreviation table and, if it is not there, in the LOGO system's abbreviation table. If we find the string listed in either table, it is effectively replaced in the line by the thing it abbreviates, and that part of the line is re-parsed.

If it is not an abbreviation, it must be a command, operation, or procedure name or a noise word. We first look for it in a table of LOGO commands and operations. (Noise words are listed with the LOGO commands.) If it is there, we create a list element in the list we are building that identifies the command or operation by its position in the command table. If it is not there, it is treated as the name of a user procedure. It is looked up in the user procedure table; if it is not already listed there, it is added to the table. Then a list element is generated identifying this user procedure by its position in the user procedure name table. The parsing process is performed by the LOGO routine PARSE. A flow diagram detailing the operation of PARSE is included in Appendix A.

There is an equivalent method of checking an input element in the form of a procedure name which takes more space but less time. The tables for built-in names and abbreviations are merged and

separated into subtables for each letter of the alphabet. If the element is not a user abbreviation, it is then checked against the members of the subtable for its initial letter. The search of this subtable, even if it fails, will take less time than the average successful search of the whole table which covers an average of half of the approximately 80 entries. Also, there are about 40 abbreviations. If the element is a procedure name and we search the abbreviation table first and then half of the built-in name table, 80 elements on the average are searched. On the other hand, the average subtable has a total length of five or six with a maximum of 15. Thus we achieve a considerable saving in time for processing many list elements. The extra space comes in the form of the table of subtables and the terminators for each of the subtables.

3. Executing a Parsed Line

LOGO is a procedural language for manipulating string expressions. An expression can be any one of the following:

1) A literal.
2) A name.
3) The name of an operation, followed by a specified number of expressions.

The value of the expression is, in each of these cases:

1) The literal itself.
2) The LOGO thing named.
3) The output of the specified operation.

Note that this definition for expression is recursive. The main LOGO routine for evaluating expressions specified by a parsed list is EXECUTE.

EXECUTE steps through a list of parsed elements and performs the indicated commands, operations, and procedures. The parsed list consists of five different types of elements. These are:

1. Names of LOGO commands and operations; noise words; and parentheses.
2. Names of user-defined procedures.
3. Literals.
4. Names
5. Comments.

EXECUTE processes a complete parsed list. It expects a parsed list corresponding to a LOGO instruction line (other than an empty line or one consisting entirely of comments) to be of the form: command name followed by a fixed number of expressions, its inputs. If this is not so, EXECUTE will generate an error comment after attempting to execute the line.

EXECUTE uses two stacks. One, the string stack, or S-stack for short, is used for accumulating all string pointers to inputs of procedures, commands, and operations not yet executed. The other, the procedure stack, or P-stack, is used for holding the names of procedures, commands, and operations encountered but not yet executed, the number of associated inputs not yet accumulated on the string stack, and associated information, other than string pointers, that is relevant to the execution of procedures, commands, and operations.

EXECUTE is entered with a pointer to the beginning of the parsed list to be executed. If the list has any leading comments, EXECUTE steps past them. If the line is now finished, EXECUTE exits. Otherwise, EXECUTE takes the next element from the list and dispatches on its type -- i.e., it performs a subroutine associated with the element type, as follows.

If the list element is a noise word, a parenthesis, or the name of a built-in command or operation (we will henceforth call these entities built-in functions), the list element, and the number of inputs it currently requires, are placed on the P-stack. Initially the number of inputs required by the built-in function is the total number specified for that function. The number is reduced by one each time an input is processed. If the number of inputs remaining to be processed is non-zero, then EXECUTE takes the next element and dispatches on it. If the number of inputs remaining to be processed is zero, EXECUTE removes the last built-in function identifier from the P-stack and goes to the subroutine referenced by this identifier.

If the list element is a literal, the string pointer for this literal is placed on the S-stack. The literal will be used as an input for the last function that was placed on the P-stack. EXECUTE decrements the number of inputs still left to collect for this function and proceeds as above when this number becomes zero, i.e., when all inputs have been collected.

If the list element is a name, the pointer to its current value (a LOGO word or sentence) is placed on the S-stack and EXECUTE then proceeds exactly as with literals.

If the list element is a comment, EXECUTE steps past it to process the next element on the line.

If there is no list element remaining, and EXECUTE has elements left on the P-stack, it generates the error message "SOMETHING MISSING" to indicate that it has not found all the inputs it needs.

If the list element is the name of a user-defined procedure, the element is placed on the P-stack. The number of inputs needed to execute the procedure is determined from the procedure definition. This number and the identifier for the LOGO routine PROCEDURE, which processes user-defined procedures, are placed on the P-stack. The user procedure is then handled by EXECUTE just as a built-in function.

Built-in functions are either *operations* or *commands*. Operations generate an output; commands do not generate any output. When EXECUTE dispatches on a particular *operation*, the corresponding subroutine removes the appropriate inputs from the S-stack and uses them to perform the operation. When the execution of the operation is completed, the subroutine puts its generated output string onto the S-stack and returns to EXECUTE which treats this new entity just as it does any literal encountered on the original list of elements. When the execution of a *command* is completed, however, the subroutine returns to a different place in EXECUTE to terminate the execution of the instruction line, since no other function can legally precede a command.

The following is a simple example of the operation of EXECUTE. It shows the state of the two stacks after each element is processed during the execution of the instruction line

PRINT BUTFIRST OF "I GO LOGO".

(In the column headed P-stack, each function is preceded by the number of inputs it currently needs.)

Element Processed	S-stack	P-stack	Comments
PRINT		1 PRINT	
BUTFIRST		1 PRINT 1 BUTFIRST	
OF		1 PRINT 1 BUTFIRST	
"I GO LOGO"	I GO LOGO	1 PRINT Ø BUTFIRST	
	GO LOGO	Ø PRINT	LOGO types "GO LOGO"
	empty	empty	

(Since the P-stack is empty*, EXECUTE exits.)

*Except for a marker which EXECUTE inserts on the P-stack on entry to denote the beginning of the line. After completion of a command, EXECUTE checks the stack for that marker. If the marker is on top of the stack, EXECUTE exits. If not, it gives an error printout.

The LOGO routine PROCEDURE is called by EXECUTE whenever all of the inputs to a user-defined procedure have been accumulated in the S-stack. The main functions of PROCEDURE are: (1) save the current position in the line being executed, (2) bind the inputs to the formal parameters, (3) call EXECUTE for each line of the procedure, and (when done with the procedure or after an OUTPUT command) (4) unbind the formal names and restore the old state of EXECUTE.

Since the procedure being called may alter the state of the compiled code, the current position must be saved relative to

the beginning of that line, along with its line number and the procedure it's in. Also, since the line itself may be altered, a version number or a line sequence number must be saved so that changes to that line can be determined. Further, the state of the truth flag must be saved, and in order to be able to unwind the pushdown stacks, if necessary, the depth of pushdown the last time through must also be saved and the current position recorded.

Binding inputs is a two-stage process. The old values associated with the formal names must be saved on a pushdown stack and the new values must be taken from the S-stack and associated with the names. There is a potential conflict since the logical place to save the old values for the formal names is also the S-stack. This difficulty only arises if the stack mechanism allows one to use only the last thing pushed. However, in virtually all existing machines one can use data other than the last-in datum on the stack. This process thus reduces to an exchange of the old values associated with the names with the corresponding number of things on the S-stack.

The following example shows the operation of EXECUTE in processing an instruction line which includes calls to user procedures. The instruction line is:

```
PRINT BUTFIRST OF BACKWORD OF "HELLO" AND STRIP OF "GOODBYE"
```

BACKWORD and STRIP are user-defined procedures. The procedure BACKWORD is defined as follows.

```
TO BACKWORD /A/ AND /B/
1 PRINT /A/
2 PRINT /B/
3 OUTPUT WORD OF /B/ AND /A/
END
```

STRIP is defined as fcllows.

```
TO STRIP /ANYTHING/
1 OUTPUT BUTFIRST OF BUTLAST OF /ANYTHING/
END
```

Element	S-stack	P-stack	Comments
PRINT		1 PRINT	
BUTFIRST		1 PRINT 1 BUTFIRST	
OF		1 PRINT 1 BUTFIRST	
BACKWORD		1 PRINT 1 BUTFIRST BACKWORD 2 PROCEDURE	
OF		1 PRINT 1 BUTFIRST BACKWORD 2 PROCEDURE	
"HELLO"	HELLO	1 PRINT 1 BUTFIRST BACKWORD 1 PROCEDURE	
AND	HELLO	1 PRINT 1 BUTFIRST BACKWORD 1 PROCEDURE	
STRIP	HELLO	1 PRINT 1 BUTFIRST BACKWORD 1 PROCEDURE STRIP 1 PROCEDURE	
OF	HELLO	1 PRINT 1 BUTFIRST BACKWORD 1 PROCEDURE STRIP 1 PROCEDURE	

Element	S-stack	P-stack	Comments
"GOODBYE"	HELLO GOODBYE	1 PRINT 1 BUTFIRST BACKWORD 1 PROCEDURE STRIP Ø PROCEDURE	
	HELLO	1 PRINT 1 BUTFIRST BACKWORD 1 PROCEDURE	PROCEDURE has bound "GOODBYE" to /ANYTHING/ and will now take instruction lines from STRIP as it calls EXECUTE
OUTPUT	HELLO	1 PRINT 1 BUTFIRST BACKWORD 1 PROCEDURE return to PROCEDURE 1 OUTPUT	
BUTFIRST	HELLO	1 PRINT 1 BUTFIRST BACKWORD 1 PROCEDURE return to PROCEDURE 1 OUTPUT 1 BUTFIRST	
OF	HELLO	1 PRINT 1 BUTFIRST BACKWORD 1 PROCEDURE return to PROCEDURE 1 OUTPUT 1 BUTFIRST	
BUTLAST	HELLO	1 PRINT 1 BUTFIRST BACKWORD 1 PROCEDURE return to PROCEDURE 1 OUTPUT 1 BUTFIRST 1 BUTLAST	

Element	S-stack	P-stack	Comments
OF	HELLO	1 PRINT 1 BUTFIRST BACKWORD 1 PROCEDURE return to PROCEDURE 1 OUTPUT 1 BUTFIRST 1 BUTLAST	
/ANYTHING/	HELLO GOODBYE	1 PRINT 1 BUTFIRST BACKWORD 1 PROCEDURE return to PROCEDURE 1 OUTPUT 1 BUTFIRST Ø BUTLAST	
	HELLO GOODBY	1 PRINT 1 BUTFIRST BACKWORD 1 PROCEDURE return to PROCEDURE 1 OUTPUT Ø BUTFIRST	
	HELLO OODBY	1 PRINT 1 BUTFIRST BACKWORD 1 PROCEDURE return to PROCEDURE Ø OUTPUT	OUTPUT is now called. Its effect is to terminate the execution of PROCEDURE by removing the return to PROCEDURE from the P-stack
	HELLO OODBY	1 PRINT 1 BUTFIRST BACKWORD Ø PROCEDURE	
		1 PRINT 1 BUTFIRST	PROCEDURE has bound "HELLO" to /A/ and "OODBY" to /B/
PRINT		1 PRINT 1 BUTFIRST return to PROCEDURE 1 PRINT	

Element	S-stack	P-stack	Comments
/A/	HELLO	1 PRINT 1 BUTFIRST return to PROCEDURE ø PRINT	LOGO types "HELLO"
PRINT		1 PRINT 1 BUTFIRST return to PROCEDURE 1 PRINT	
/B/	OODBY	1 PRINT 1 BUTFIRST return to PROCEDURE ø PRINT	LOGO types "OODBY"
OUTPUT		1 PRINT 1 BUTFIRST return to PROCEDURE 1 OUTPUT	
WORD		1 PRINT 1 BUTFIRST return to PROCEDURE 1 OUTPUT 2 WORD	
OF		1 PRINT 1 BUTFIRST return to PROCEDURE 1 OUTPUT 2 WORD	
/B/	OODBY	1 PRINT 1 BUTFIRST return to PROCEDURE 1 OUTPUT 1 WORD	
AND	OODBY	1 PRINT 1 BUTFIRST return to PROCEDURE 1 OUTPUT 1 WORD	
/A/	OODBY HELLO	1 PRINT 1 BUTFIRST return to PROCEDURE 1 OUTPUT ø WORD	

Element	S-stack	P-stack	Comments
	OODBYHELLO	1 PRINT 1 BUTFIRST return to PROCEDURE Ø OUTPUT	
	OODBYHELLO	1 PRINT Ø BUTFIRST	
	ODBYHELLO	Ø PRINT	LOGO types "ODBYHELLO"

- - - - - - - - - - -

At this point, both stacks are empty so the execution is completed and EXECUTE exits.

As the example shows, during the evaluation of a LOGO expression (or the execution of a LOGO instruction line) many inputs may be accumulated and many operations may be called before the expression is reduced to a single value.

It can be shown that one pushdown stack is sufficient to accomplish the evaluation of any expression. It is also true that the use of two stacks is more convenient for performing auxiliary functions such as garbage collection (see Section 6).

There are at least two apparently different, but equivalent, encodings of EXECUTE:

1) when the EXECUTE routine encounters an operation, it accumulates the appropriate number of inputs before invoking the operation,

2) when EXECUTE encounters an operation, the operation is invoked immediately, and it in turn calls EXECUTE n times, where n is the number of its inputs.

In the first instance, what appears on the stack are the inputs already found, the names of the operations to be called, and the number of inputs remaining to be accumulated. In the second instance, what appears on the stack are the outputs already accumulated and the place to go after the next output is put onto the stack.

Almost any encoding of EXECUTE is sufficient to execute an instruction line with no errors. A good encoding is one that not only performs the embedded functions and procedures correctly and efficiently, but also detects all possible user errors and preserves lots of relevant information for generating insightful error comments.

The following errors must be detected by EXECUTE.

THERE ARE *n* INPUTS MISSING FOR *x*. End of line reached while looking for an input for *x*.

x IS EXTRA. Elements are still unused after a command completes its action.

x CANNOT BE USED AS AN INPUT. IT DOES NOT OUTPUT. There is something left on the line to the left of the command *x*. (This could have been detected at COMPILE time for built-in commands, but not for user procedures which do not produce an output. So, for symmetry, the detection of this error is deferred until run time.)

THERE IS NO COMMAND ON THIS LINE. This also could have been detected at COMPILE time for built-in commands but not for user procedures, so it is also deferred.

4. Internal Representation of Data

Along with specifications for the several LOGO commands and operations, the preceding descriptions of the PARSE and EXECUTE routines are, in principle, sufficient for designing a LOGO processor. The rest of this report deals primarily with considerations relating to designing an *efficient* processor. The difference between a relatively efficient implementation of LOGO and one with obvious inefficiencies can be dramatic in terms of the speed of service to an individual user and/or the total number of users that can be simultaneously served on a given multi-user system.

A major part of the time spent in LOGO processing is taken with parsing and executing instructions. PARSE can be a costly process because it involves scanning each of the individual characters on an instruction line, one at a time, and because it requires a great deal of searching of tables to determine the meanings of the various names encountered in the line. EXECUTE can be a costly process because it may require a great deal of referencing to process a parsed list.

We can reduce the time spent in parsing lines mainly by performing PARSE just once for each line, instead of parsing a line each time it is referenced. This leads to an implementation that is efficient in space as well as time usage. We need to save only the parsed list of elements not the original text string of the line, and the parsed list need not require appreciably more space -- and often requires less space -- than the original string.

The main way to reduce the time spent in executing parsed lines is to eliminate all searches for the values of elements referenced during EXECUTE. Thus, our strategy here is to structure the data used during EXECUTE so as to enable elements to be accessed *directly*. Figure 1 is a schematic diagram showing how the various types of data used during EXECUTE are structured to permit direct referencing.

EXECUTE operates on a parsed list. A list element must contain two pieces of information. One is an indicator of the element type and the other is a pointer to the value of the element. In the case of literals and comments the pointer is a pointer to the text of the element in the string storage. In all the rest (names, names of built-in functions, names of user procedures), the pointer is a pointer into one of three tables. The tables are all of the same form. Each entry in each table contains two pieces of information. One is a pointer to the name of the element and the other is a pointer to the value.

The table for built-in names contains an entry for every word known to LOGO. Since these names are all known in advance, the names do not occur in the standard string storage but in LOGO's permanent storage. The tables for names and procedure names contain an entry for each and every name of that type that has ever been encountered in parsing of input lines.

For names, the value is a text in the string storage. For built-in functions, the value is a subroutine which produces the desired effect. For user procedure names, the value is a list of entries in another table, the table of procedure directories. Each entry in this list, like entries in all other lists, has two pieces of information. The first entry of each list is

Elements which make up a Parsed List

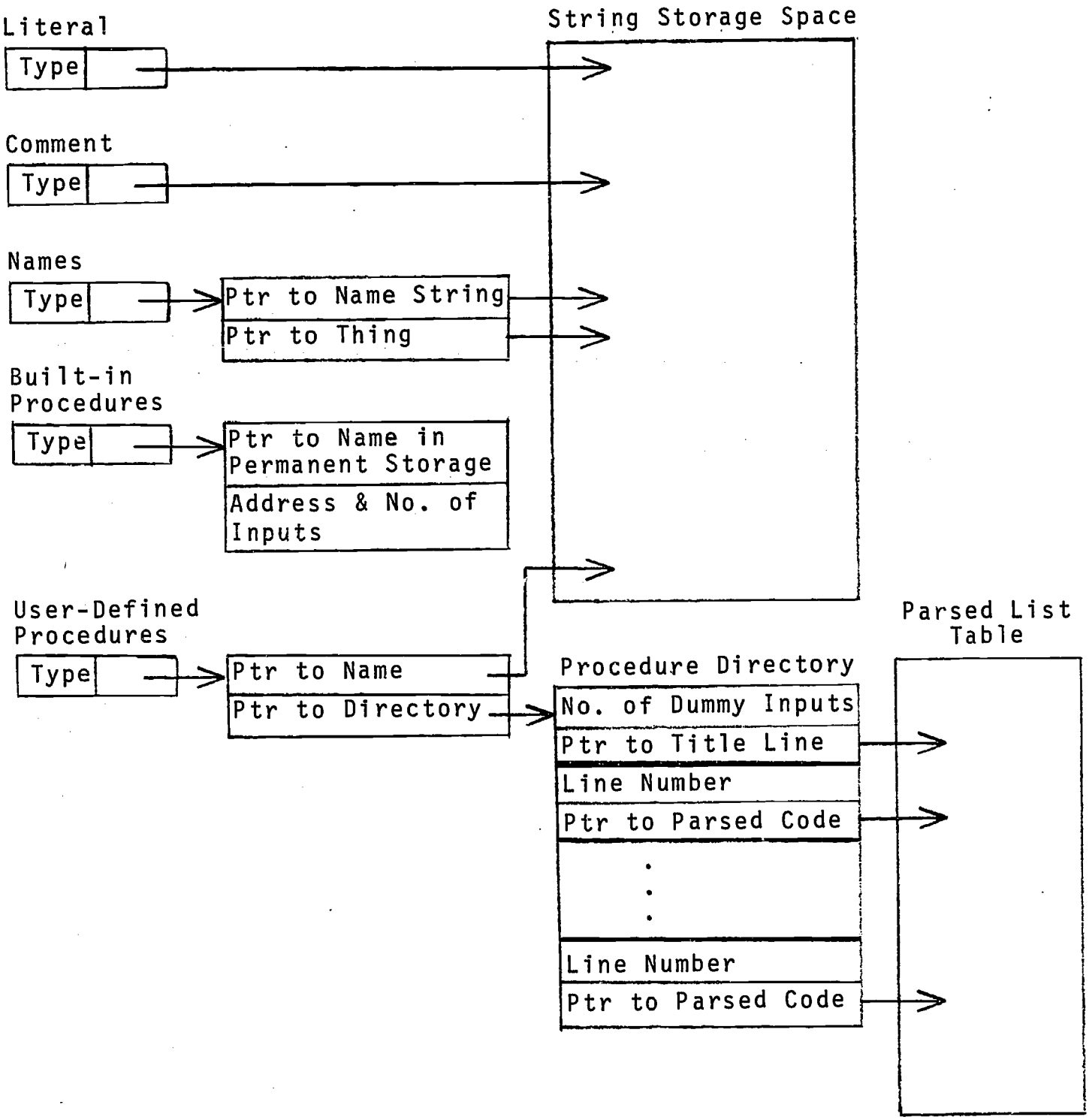


Fig. 1. Structure of Data Used During EXECUTE.

special. It contains the number of inputs to the procedure and a pointer to the parsed list of the title line of that procedure. The rest of the entries in each list contain a line number and a pointer to the parsed list of that instruction line. These entries are kept in ascending line number order. This form of procedure directory was primarily designed to make referencing as direct as possible.

All data in LOGO, including numbers, are strings. Strings can be of any length. To manipulate number strings, one can either provide number-string conversion and arithmetic for arbitrarily long integers or else perform the arithmetic operations directly on the character strings themselves. In PDP-10 LOGO we chose the latter course.

Since integer arithmetic is most naturally performed least significant digit first, but strings are most easily accessed most significant digit first, PDP-10 LOGO reverses the input strings before arithmetic operations and then reverses the output strings afterwards. In division, which is done by repetitive subtraction and shifting, starting with a subtraction of a like number of digits in both the divisor and the dividend, both the quotient and the remainder are generated in the same process.

We asserted at the beginning of this chapter that the parsed form of LOGO instructions, along with the procedure directories and variable tables, might take up less space than the original text. We can show that for a program of reasonable complexity a storage compression ratio of three to two or even more is feasible. For the illustration on page 10, for example, the thirty characters reduce to twenty; and for the instruction line along with the two procedure definitions on pages 12 and 13, we get a ratio of 209 to 143, i.e., approximately 3 to 2.

5. Storage Allocation

The previous section on data structures did not discuss the sizes of the various tables or how these might change across time. There is no a priori selection of table sizes that will satisfy the needs of all users. Different programs and programming styles will generate different requirements for table sizes of the various data types. Thus, it is desirable that the only restriction on the size of tables should be the unavoidable one that the total amount of data in all the tables cannot exceed the maximum amount of space available to the user.

We therefore expect to vary the sizes of tables dynamically according to the user's needs. Any set of initial table sizes will suffice. If any type of data grows to exceed the size of its table, the size of that table will be expanded. All the data following that table will then be moved up and the pointers to these data appropriately adjusted. In general, with LOGO, we should expect this to happen routinely.

When one table is expanded, the tables after it are moved only as a whole. Data do not change their relative positions within each table. Thus, if all pointers into tables are made relative to the beginning of their tables, when a table expansion occurs, only the pointers to the beginnings of the tables following the expanded table need be modified, but not pointers to individual data within those tables.

If the process of expanding a table will cause the total amount of allocated space to exceed some bound, then any unused allocations in the various tables will be recovered to permit the necessary expansion. The bound might be the top of core or the top of a quantum of storage allocation in a multiple-user variable allocation scheme.

When starting a LOGO session, no user data have been created, and there is no reason to assign more than a minimal amount of space until such data are generated.

Many charging algorithms penalize users with large core allocations. The penalties are not necessarily monetary but may be in degraded overall system performance. Moreover, if the scarcest resource in the system is memory space rather than processor time, and this is typically the case for interactive systems, it is worthwhile to spend some percentage of time keeping the utilized memory to a minimum.

Thus, a means of eliminating space for data which are no longer referenced may be of considerable practical value. This kind of process is called *garbage collection*.

6. Garbage Collection

Thus far, we have discussed only the generation and storage of data. Whenever LOGO runs out of memory for data storage there are two possible ways to get more. One is to request more from the host system. This way fails if the memory allocation is fixed or if it is at the maximum. The other way is to eliminate any data which are no longer referenced. One situation in which data become unnecessary is when an instruction line is replaced. The space occupied by the superseded parsed list is no longer needed and should be recovered at that time. When an instruction line is erased, the space for both the parsed list and the procedure directory entry for that line should be recovered. To recover the space occupied by a parsed list, all parsed lists

following the superseded list are copied down a number of cells equal to the length of that list, then, all the pointers to parsed lists in all procedure directories which point to lists which have been moved are decremented by the amount the lists have been moved.

If the procedure directory is not the last one in the procedure directory table when it expands or contracts, then all of the directories above the one being modified must be moved for each line that is added or deleted. But typically, when a procedure is edited, more than one line of the procedure is added or deleted at a time. Thus, when the editing of a procedure is initiated, that procedure directory is moved to the end of the table of procedure directories and all subsequent modifications to that procedure directory affect only that directory, and not those which previously may have followed it.

Names and procedure names have a permanent entry assigned to them in their respective name tables. It is possible that some of these entries may become unnecessary due to the deletion of all references to them in the parsed code. These can be recovered by the following garbage collection technique.

(1) Search for all references to names and procedure names in the parsed code (the only place they can occur) and mark the table entries that are referenced.

(2) Search each table from the bottom for unreferenced entries. When one is found, search the table from the top for a referenced entry. If one is found above the unreferenced one, move the referenced entry down and replace it with a pointer to where it was moved. This process is terminated when there are no referenced entries above unreferenced ones.

(3) Search the parsed code again for name and procedure name elements. Check each one to see if it points at a pointer instead of an entry. If it does, update the element to point at the entry rather than the pointer.

Thus far we have only considered the process of garbage collection for non-string data. We have not yet discussed the form or handling of strings, other than acknowledging their existence.

All strings are stored in the string table in order to localize the special processing required for strings. Strings present special problems because they can be arbitrarily long. Moreover a string is generated whenever a LOGO operation is performed and there is no way of knowing whether or not it will be saved and, if so, for how long a period. Thus, the storage allocation and garbage collection of strings are not the same as with other data.

Depending on the particular form used for representing strings, the amount of storage needed for them, and the amount of time needed to garbage collect them, can be very different. We shall consider three distinct ways of representing strings. The first uses the least space, but requires the most time for garbage collection; the second takes the most space of the three representations, and requires more time than the others for storage and retrieval, but less time for garbage collection; the third usually takes less space than the second without requiring significantly more time for garbage collection.

In the first representation, strings consist of characters starting at a word boundary and terminating with an EOM. In the PDP-10 implementation, characters are seven bits long and stored five to the word. This leaves the least significant bit of any string word free for marking strings to be saved.

The space allocated for string storage is located after all other storage areas. This allows for the use of the upper bound of all assigned memory to be used as the upper bound on the string storage. Then the test for no more room in the string space can be the occurrence of a memory bound violation on a store character instruction instead of a pointer compare before every store character.

The garbage collection procedure consists of two parts. First, all data types that use string pointers are searched (they are: names and their things, abbreviation names and their things, the S-stack, procedure names, and literal and comment references in the parsed code). For each string pointer found, the corresponding string must be marked for saving. A string is marked by setting the spare bit of its first word.

Second, the string storage is searched word-by-word until it finds one with the spare bit set. When one is found, all possible pointers are scanned for those that point to this string (there may be more than one) and modified to point where the string will be located after it is moved down over the garbage strings. Then this string is unmarked and is copied word-by-word and the location following its last cell will become the first cell available for the next string. This process is repeated until all the strings have been passed over.

This method requires N+1 full scans over the string pointers (where N is the number of strings being saved), and might be reasonable for a system that has little or no available memory but a great deal of available time. A machine with little memory cannot afford space for any kind of overhead words in

strings. However, since so little memory is available, there cannot be enough strings to make the garbage collection time prohibitive.

The second way of representing and storing strings is to list structure them. This has the following advantages: (1) good strings do not have to be moved in order to fully utilize the recovered space, (2) the pointer spaces have to be passed over only once (marking pass) because the pointers do not have to be altered since the strings do not move.

The marking pass consists of searching all the pointer lists for string pointers and marking all the cells of all the good strings. The collection pass consists of passing over the string space searching for unmarked cells and chaining them to each other and also unmarking those that had been marked. This method affords considerable savings in time for the garbage collection because the pointer spaces and the strings have to be scanned only once each.

The disadvantages of this representation are: (1) the storage and retrieval of characters in strings stored with pointers embedded in them must be done by subroutine, causing some added processing time even for a machine with good character handling instructions; (2) the amount of storage allocated to strings cannot be reduced without further manipulation similar to that used with the first type of string representation; (3) the storage efficiency is less than with the first type of string representation.

A string cell in this representation is a contiguous group of storage locations. Each cell contains a pointer to the next cell

of the string or a termination pointer. The rest of the cell space is occupied by the characters comprising the string. The storage efficiency obtainable with this type of string storage varies according to the choice of cell size. The maximum efficiency possible using a single word for a cell is 40%; for a cell size of two words, the maximum efficiency goes up to 70%; for a four word cell size it can approach 90%. These figures assume that all available space for characters in the cell is used, and that the chain pointer is 18 bits long, leaving space for two characters in the same 36-bit word.

The process of reducing the amount of space allocated to strings in the list-structured representation is the same as the fixed garbage collection described on pages 25 and 26.

Note that having a fixed location for temporarily keeping pointer information while strings are being moved saves some scanning of the pointer list. With that in mind, we next propose a representation which incorporates an overhead word for each string. This word can be in a separate table or it can be interleaved with the strings. Let us compare the space utilization for strings of various lengths stored with one overhead word per string versus list-structured string cells of various sizes. For the one word list-structured cell:

String Length (characters)	List-Structure (words)	Contiguous String plus one word (words)
1-2	1	2
3-4	2	2
5-6	3	3
7-8	4	3
9-10	5	3

The only case where there is any saving with one word cells is in strings of one and two characters.

For two word cells:

1-5	2	2
6-7	2	3
8-10	4	3
11-14	4	4
15	6	4
16-20	6	5

For four word cells:

1-5	4	2
6-10	4	3
11-15	4	4
16-17	4	5
18-20	8	5
21-25	8	6

In each case, as the string gets longer, the amount of space required with the list-structured representation gets progressively worse by comparison to the new scheme.

If the overhead word for a string is kept in a separate table, it can be used in any of a number of ways. One way is to use it as a fixed location which contains the location of the text of the string and string pointers in turn point to it. Garbage collection for this configuration consists of scanning the lists for string pointers to good strings. Marking a string is done by putting the first word of the text into this fixed location and pointing to the table where the text used to be. At the end of the marking pass the string pointers and the strings both point into this table. The string compression pass consists of searching for marked strings. When one is found, the first word of text is retrieved from the table and replaced by the new location of the string. The entire string is then moved to the first available slots for good text. It may be necessary to compress the table as well. Compressing the table is done in a manner equivalent to that used for compressing the name tables as described above. The table in this instance is, in effect, an indirect address for the strings.

In another approach, the extra word is used only during garbage collection. One has to insure that there are at least as many words in the table as there are strings. The marking pass in this case consists of putting the first word of the string into the table and making both the string itself and the string pointer point into the table. The string compression pass is the same as described above, but this time there always needs to be a third pass, to replace the relocated string pointers with the contents of the table address pointed to by the relocated string pointer. Here the table acts as a fixed address for the string only while it is being moved.

Both of these ways of using the overhead word suffer in that the allocation of space for the extra table is not exactly in step

with the generation of strings. If all allocation areas are contiguous and the strings are stored at the end, the strings need to be moved each time the size of the extra table is increased. Also, the string storage area may fill up while there is room still left in the table, and this is not optimal. Finally, strings still have to be searched word-by-word and moved a word at a time.

In yet another storage technique, the word of overhead for each string is put at the head of the string itself. Half of this word is used to contain the length of the string. This length information has a number of uses: (1) when comparing two strings, if the two strings are of differing lengths, they will not match on the first word of the comparison. This may be a significant saving in comparison time. (2) Since one knows in advance where the end of the string is, during the string compression a block transfer can be used instead of a word-by-word copy, because the source, the destination, and the length of the string are all known.

The other half of the word is used during garbage collection by making it the head of a linked list of all of the references to that string. During the marking pass this half word is used to denote that the string is to be saved by making it point at the pointer that refers to this string. Thus, during the compression pass, the pointers to the strings can be updated without searching, by using this back pointer to update the string pointers. Since a string can be pointed to by more than one pointer, it is necessary to treat the back pointer as a pointer list. We do this by putting the current contents of the marking half of the overhead word into the address part of the string pointer, and putting the address of that string pointer into the marking half of the overhead word. Then, after the marking pass is complete, one can find

all references to a string by tracing back through the pointer list starting at the string itself and ending with the terminator that was in the overhead word before the marking pass was initiated. Another advantage of having the string length stored with the string is that the search for marked strings is as small as possible because it can be limited to the first word of each string.

The above discussion on storage allocation assumes that the data of the various types are kept in separate tables, varying in internal structures, with separate garbage collection techniques for each. An alternate method of data storage is to consider all data to be of the same form, the one relegated only to strings in all the preceding discussion, i.e., one word of overhead and any number of words of data. Lines of compiled code can be handled this way, as can individual procedure directories, the list of all procedure names, the list of all names, and the list of abbreviations. In fact, all data in LOGO, except for pushdown lists, can be handled in this manner.

Even a sentence can be a list of pointers to words, rather than a string of characters. This representation can lead in a straightforward way to extensions of the language to other data types and operations on these types.

Using this type of data structure, some small changes in the handling of data are required. When a list is added to and there is no room left in that list or table, instead of moving up the data following the list to make room for the expansion, the table is moved to the end of the data storage and extra room is made there.

Now that elements can contain pointers to other elements as well as data, garbage collection of the data in such a mixed structure also differs, because pointers to elements in the structure are data in other elements. The marking pass is the same as in the last method above. The second pass in the previous method, the one that does the pointer fixups and element compression, must be altered. Otherwise, if any element compression is done before the pointer fixups are complete, some of the pointers in the fixup chains may move and the chains will then become invalid. Therefore, this pass is itself broken up into two passes, the first for doing the pointer fixups, and the second for doing the storage compression once the pointers are all modified (to reflect the state of affairs after the compression would have been done).

7. Filing Requirements

A filing system is an integral part of LOGO. An essential aspect of the use of LOGO involves the elaboration and extension of previous programs. To eliminate the need for reentering the growing body of previously written programs, at each session, we provide a filing system which satisfies the following requirements.

(1) Each student should have his own catalog of saved programs (that is, his own file).

(2) He should easily be able to
   (a) see what programs are in his file or other files made available to him;
   (b) add programs to his file;
   (c) add to or otherwise modify a program in his file;
   (d) remove any program from his file;
   (e) load one or more programs from his file.

(3) A program saved in a file should contain names, abbreviations, and procedures.

(4) When a program is loaded into LOGO from a file, the effect should be the same as if the program had been entered on the terminal. Similarly, the input of a DO command should have the same effect as if it was loaded from a file or typed in at the terminal.

8. General Design Considerations

LOGO, like many programming languages, is most effectively used in an interactive environment. The LOGO system provides all the facilities, such as editing and debugging, required to give the user complete control of everything he needs to carry out a working session. He does not need to know the special and relatively complex conventions necessary to operate the equivalent facilities that may be provided by the host system.

When LOGO must be used in a system which does not provide the facilities for interactive operation, a LOGO compiler might be written in preference to a semi-compiler or an interpreter. However, the resulting code that would be compiled has, essentially, a one-to-one correspondence with the parsed elements of the semi-compiler which we have described. The central difference is that the code produced by the compiler will be a list of subroutine calls to the same routines that are dispatched to by the EXECUTE routine in the case of the semi-compiler. The compiler gains relatively little either in space or time, in typical cases, over the semi-compiler which, in turn, is superior to an interpreter both in space and time savings, as we indicated in Section 4.

In a LOGO system we need to provide a sizable amount of storage for user data, in one form or another. This storage can be split between primary memory and high-speed devices such as discs or drums. Even tape can be effectively used as secondary memory in a one-user system but a disc or drum is essentially necessary for a multi-user system.

Several questions must be asked about implementing LOGO on smaller computers.

1. How small a computer?
2. How much of LOGO is to be included?
3. What kind of environment?

LOGO inherently uses large amounts of storage. This is not a function of the implementation so much as the character of the language. LOGO is a string and/or list handling language. By their very nature, strings and the procedures that process them take much more storage than numbers and numeric procedures. Thus, on a 12-bit or 16-bit computer we think that at least 24K bytes of memory are necessary to implement a single user in-core LOGO system. Also, in implementation, space must be the prime design criterion. Thus such methods as efficient variable storage, storing strings as lists of words, storing only one copy of each word or string, etc., are all the more important. Some form of file storage should also be included, for one of the basic principles in LOGO's design is that each student builds on his past work.

Attached as Appendix C is a list of what we consider a minimal set of LOGO commands, operations, and names. A command or operation was eliminated if its use was infrequent and it could be written as a LOGO procedure, but some commands were retained that were trivial to implement.

The final question concerns the type of environment. Classroom use of LOGO will generally require a multi-user system. Such a system will necessitate at least 8K bytes more memory and certainly a high-speed swapping device. Its implementation, however, would not be too different from the implementation of a single-user system on the same machine, assuming that the user-specific data is kept separate from the LOGO system code.

There are situations where an even smaller machine might be used to implement a LOGO processor. For these, a swapping device is absolutely necessary because either the LOGO processor or the data will not entirely fit in core. There is a logical way of partitioning the sections of a LOGO processor. The set of parts are: the input and parsing; EXECUTE and the operations and commands; listing, editing, and erasing; filing; garbage collection. The operations can be separated from the commands and even from each other if absolutely necessary.

There are many possibilities for the partitioning of LOGO data. The pushdown list or lists can be segmented to secondary storage. The procedures can be broken down into single instruction line segments on secondary storage. Also, the procedure name table and the variable table can be segmented. In a situation where all of these occur, it is probably better to keep name strings in the tables and literal strings with the instruction lines and the operands themselves on the stack rather than having string pointers into a separate string or list area in an attempt to reduce the number of virtual references to retrieve a datum. The only place where pointers rather than strings would have to be kept is for the values in the name table because one wants to keep a fixed relationship between a name and its position in the name table for speed in referencing.

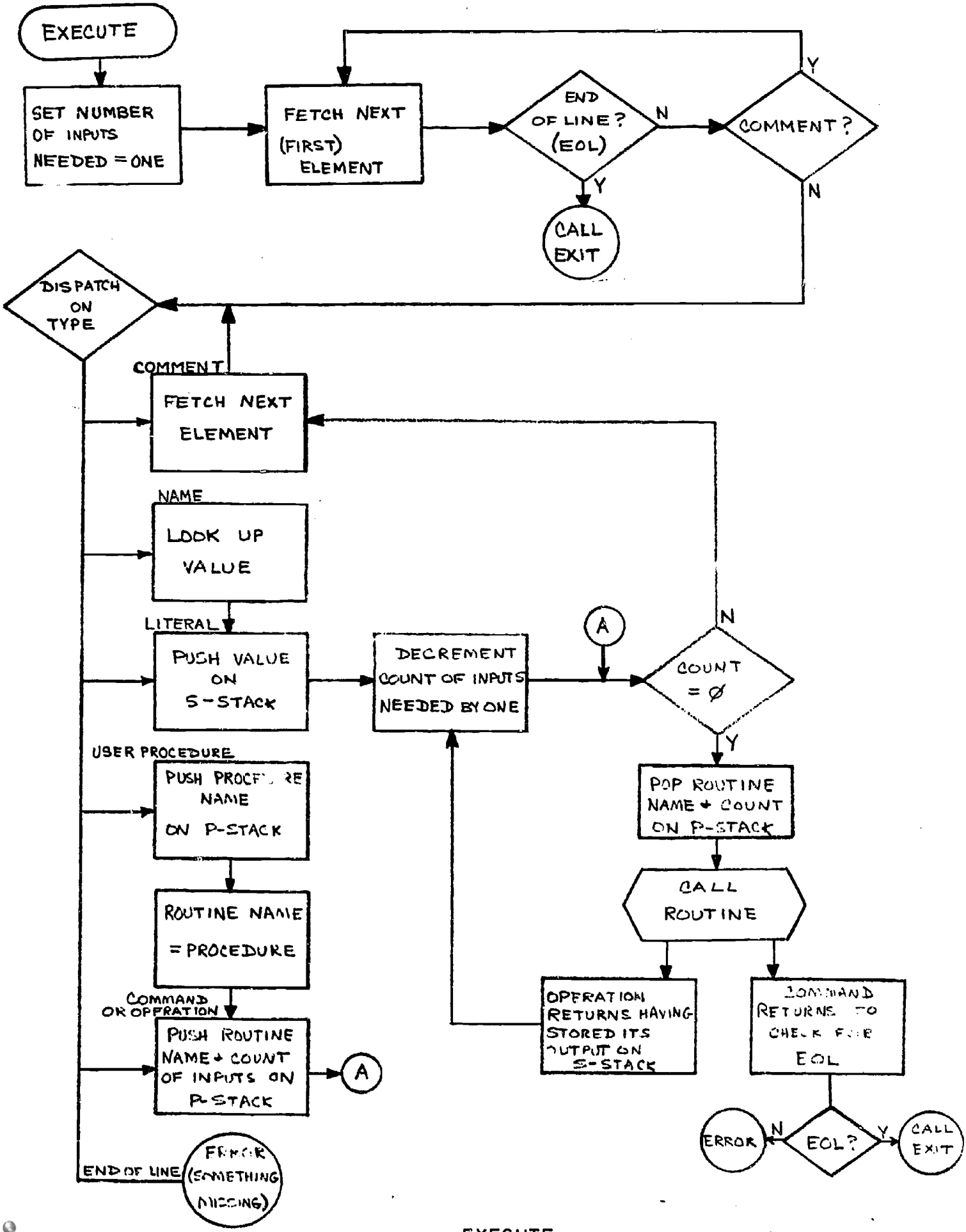
When lines of procedures are in separate segments, it is desirable to have them chained forward for fastest sequential execution and also backward for ease in editing. To keep GOTO's from getting excessively slow, it would also be advantageous to have a table of contents (or index) for the lines of the procedures. An index is also helpful for rapid retrieval of variable references from the name table and also for the procedure name table.

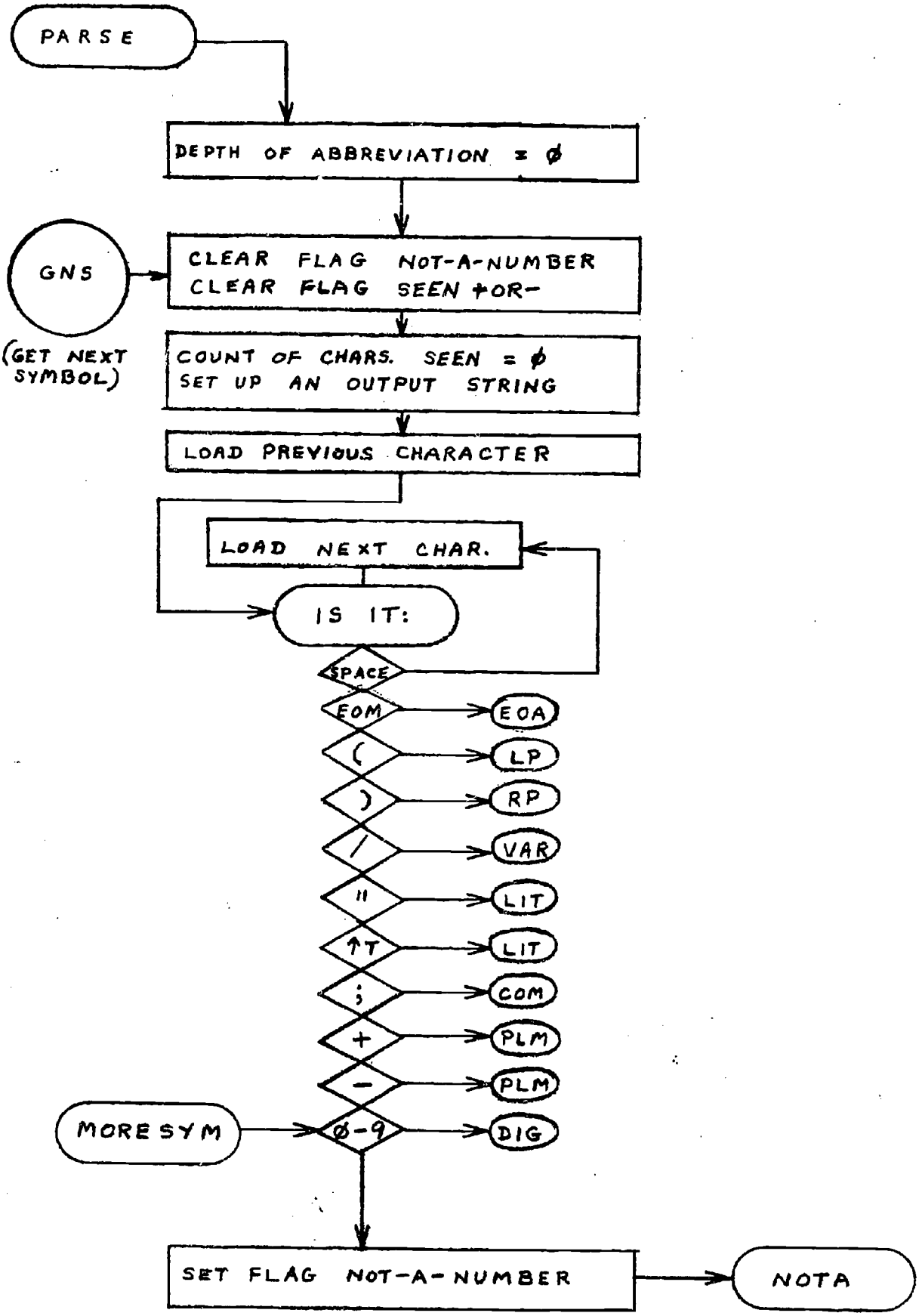
With the kinds of data partitioning described above, the amount of core needed for a single user's data can be made very small. Most of the available time will be spent waiting for data from secondary storage. One way to increase throughput is to reduce latency. However, there is no foolproof look-ahead scheme for a single user. Therefore, it would probably pay to have the minimum amount of data in core sufficient to accommodate many users at the same time, where *many* is on the order of the number of segments that can be individually read in one revolution of the secondary storage.

An example of a fairly minimal configuration for implementing a LOGO processor is a DEC PDP-8 (or a comparable machine) with 8K of memory and a DEC tape, microtape or other form of addressable tape. Half the primary memory is used for code and the other half for user data. A relatively complete LOGO implementation, realized on a DEC PDP-10 requires 4K or 36-bit words for the LOGO code; another relatively complete one, realized on a DEC PDP-1 takes 8K of 18-bit words.

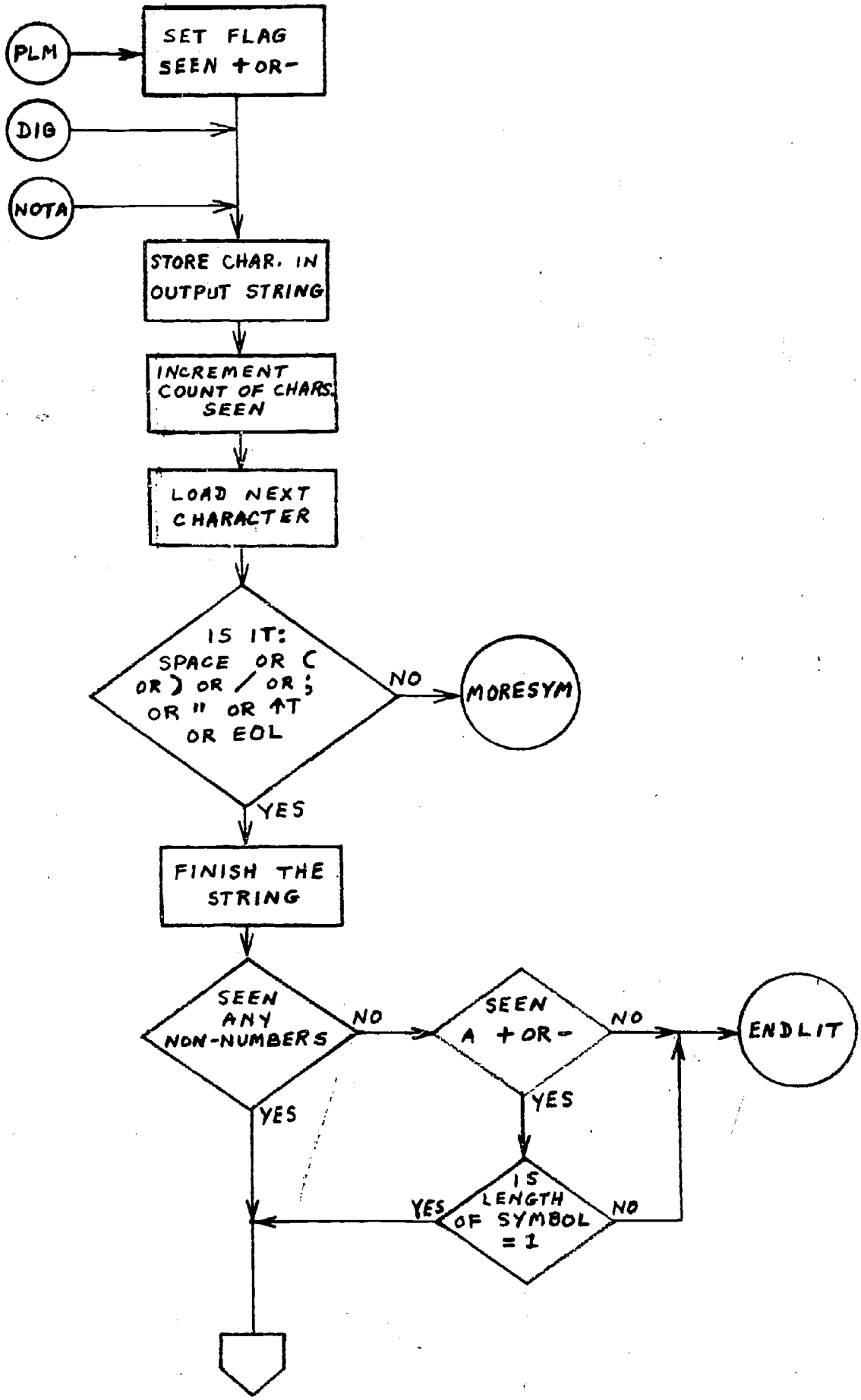
The LOGO implementation we have described is a general one. The basic algorithms used for parsing and execution would be roughly the same if one were interested in minimizing space rather than time. Even such a radical change in optimization strategy would mainly affect the internal representation: the list elements might be structured somewhat differently, e.g., different elements might have different lengths. Aside from this, the only process that would be significantly affected is garbage collection, and we have described a range of different algorithms spanning the significant options here. Essentially then, to within these variations in garbage collection algorithms, the LOGO design applies to a variety of distinctly different machines, configurations, sizes, speeds, and operating environments.

Appendix A: Flow Diagrams for
EXECUTE
PARSE
TIS (Type in String)
TO (Define Procedure)

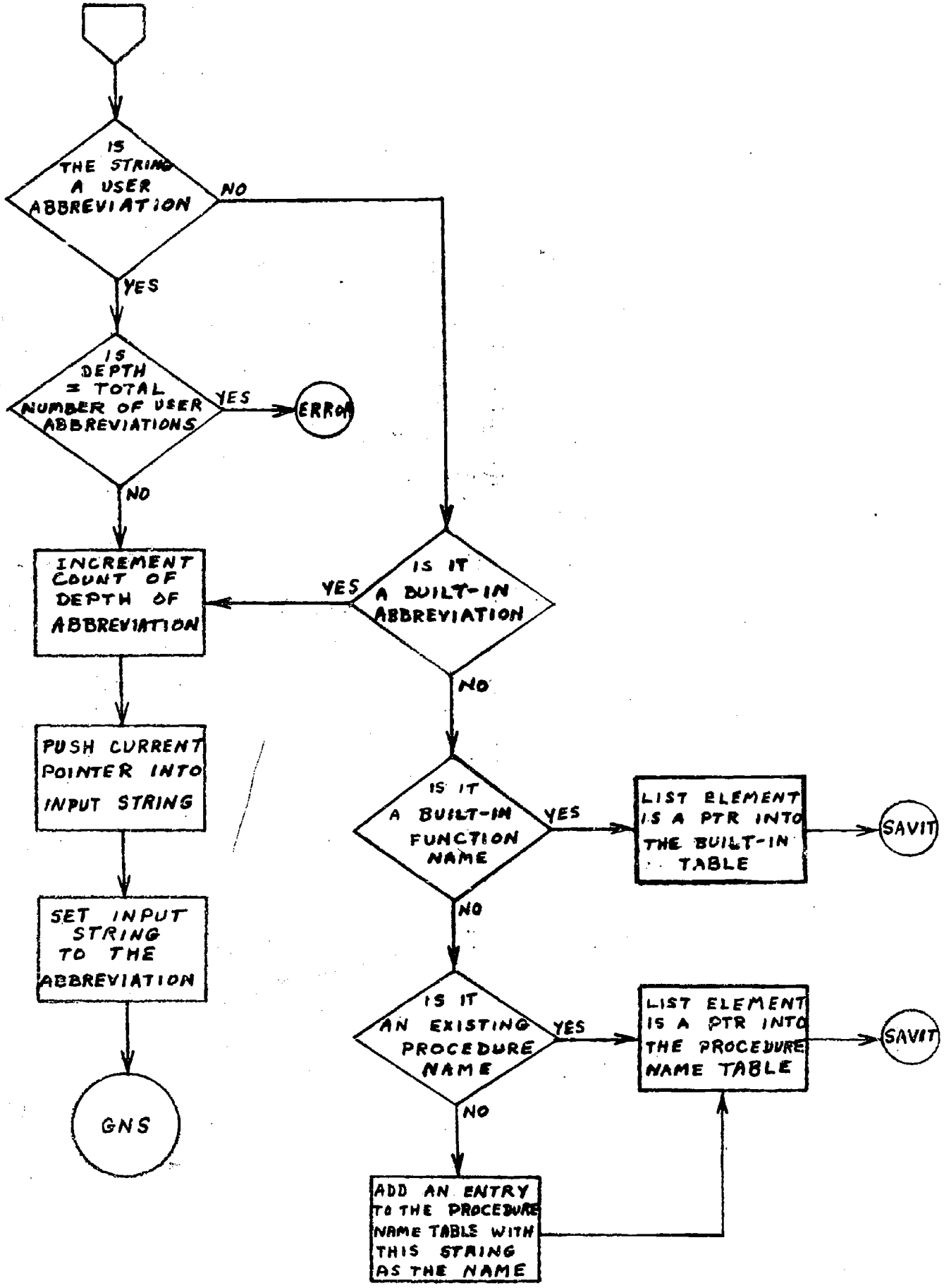

EXECUTE
SET NUMBER OF INPUTS NEEDED = ONE
FETCH NEXT (FIRST) ELEMENT
END OF LINE? (EOL)
N
Y
COMMENT?
Y
N
CALL EXIT
DISPATCH ON TYPE
COMMENT
FETCH NEXT ELEMENT
NAME
LOOK UP VALUE
LITERAL
PUSH VALUE ON S-STACK
DECREMENT COUNT OF INPUTS NEEDED BY ONE
A
COUNT = Ø
N
Y
USER PROCEDURE
PUSH PROCEDURE NAME ON P-STACK
POP ROUTINE NAME + COUNT ON P-STACK
CALL ROUTINE
ROUTINE NAME = PROCEDURE
COMMAND OR OPERATION
PUSH ROUTINE NAME + COUNT OF INPUTS ON P-STACK
A
OPERATION RETURNS HAVING STORED ITS OUTPUT ON S-STACK
COMMAND RETURNS TO CHECK FOR EOL
END OF LINE
ERROR (SOMETHING MISSING)
ERROR
N
EOL?
Y
CALL EXIT


PARSE
DEPTH OF ABBREVIATION = ∅
GNS
(GET NEXT SYMBOL)
CLEAR FLAG NOT-A-NUMBER
CLEAR FLAG SEEN +OR-
COUNT OF CHARS. SEEN = ∅
SET UP AN OUTPUT STRING
LOAD PREVIOUS CHARACTER
LOAD NEXT CHAR.
IS IT:
SPACE
EOM
EOA
(
LP
)
RP
/
VAR
"
LIT
↑T
LIT
;
COM
+
PLM
-
PLM
MORE SYM
∅-9
DIG
SET FLAG NOT-A-NUMBER
NOTA

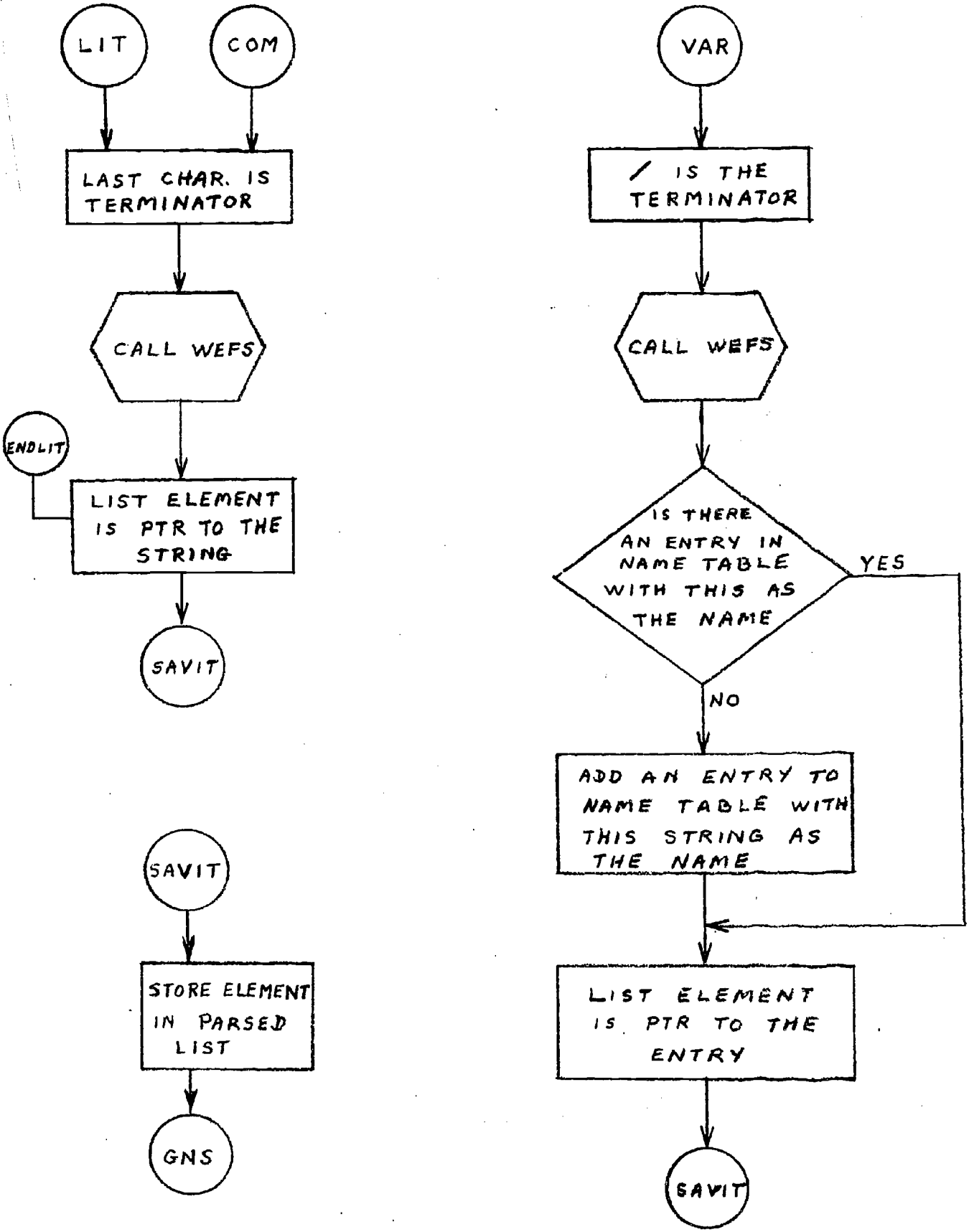
PARSE - 1


PLM
DIG
NOTA
SET FLAG SEEN + OR -
STORE CHAR. IN OUTPUT STRING
INCREMENT COUNT OF CHARS. SEEN
LOAD NEXT CHARACTER
IS IT: SPACE OR (OR) OR / OR ; OR " OR ↑T OR EOL
NO
MORESYM
YES
FINISH THE STRING
SEEN ANY NON-NUMBERS
NO
SEEN A + OR -
NO
ENDLIT
YES
YES
IS LENGTH OF SYMBOL = 1
YES
NO

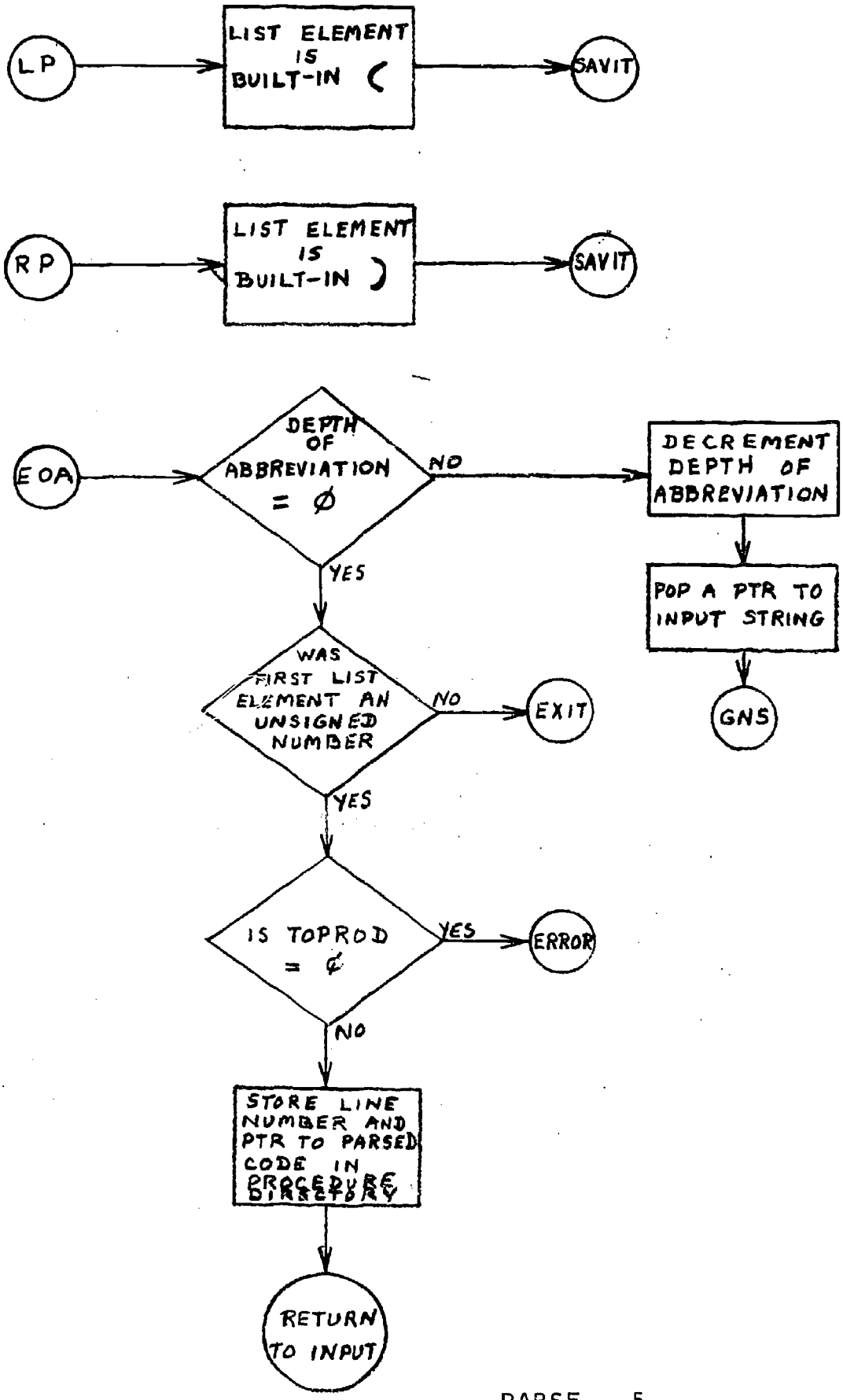
PARSE - 2

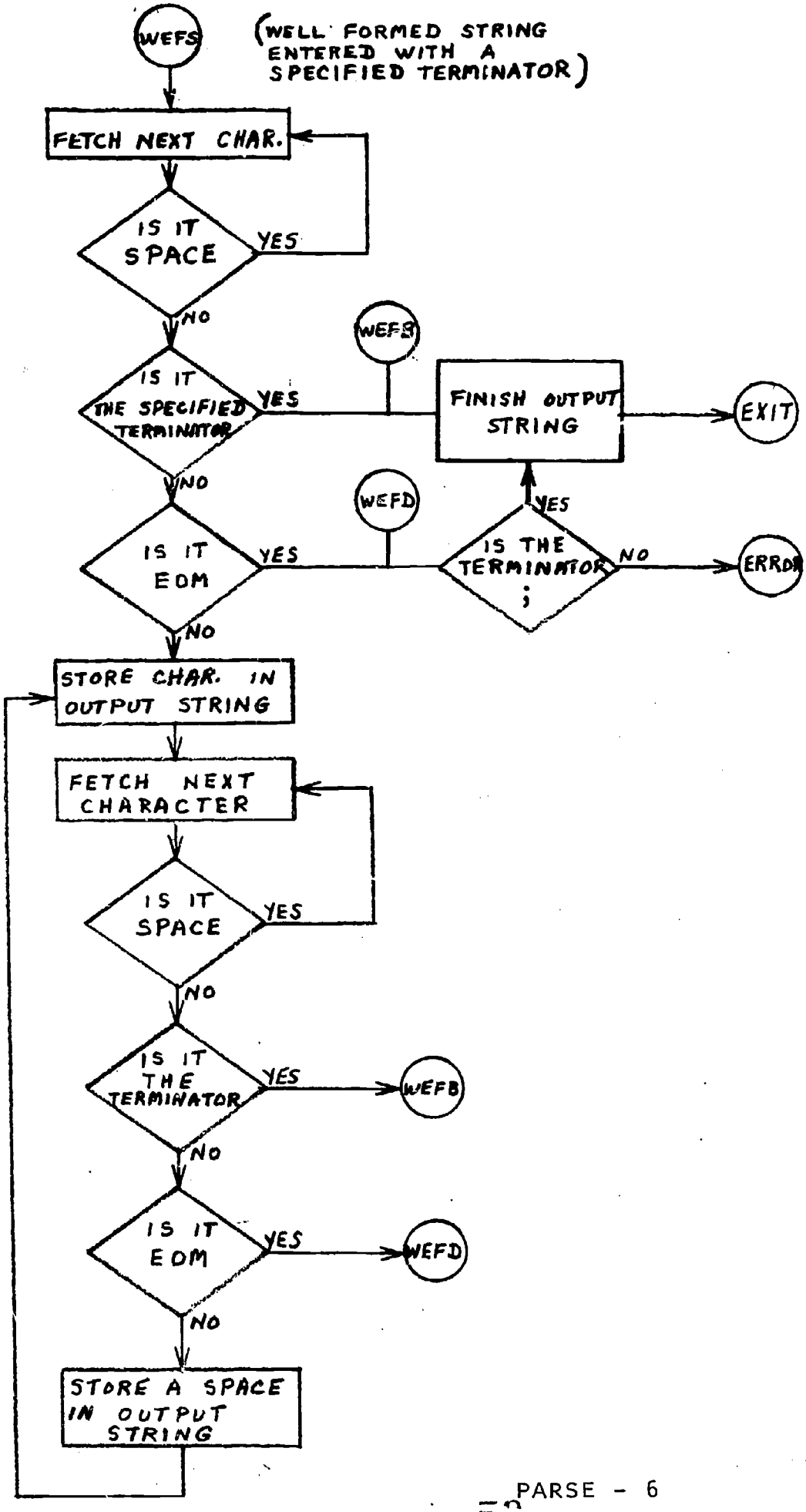

IS THE STRING A USER ABBREVIATION
NO
YES
IS DEPTH = TOTAL NUMBER OF USER ABBREVIATIONS
YES
ERROR
NO
INCREMENT COUNT OF DEPTH OF ABBREVIATION
YES
IS IT A BUILT-IN ABBREVIATION
NO
PUSH CURRENT POINTER INTO INPUT STRING
SET INPUT STRING TO THE ABBREVIATION
GNS
IS IT A BUILT-IN FUNCTION NAME
YES
LIST ELEMENT IS A PTR INTO THE BUILT-IN TABLE
SAVIT
NO
IS IT AN EXISTING PROCEDURE NAME
YES
LIST ELEMENT IS A PTR INTO THE PROCEDURE NAME TABLE
SAVIT
NO
ADD AN ENTRY TO THE PROCEDURE NAME TABLE WITH THIS STRING AS THE NAME

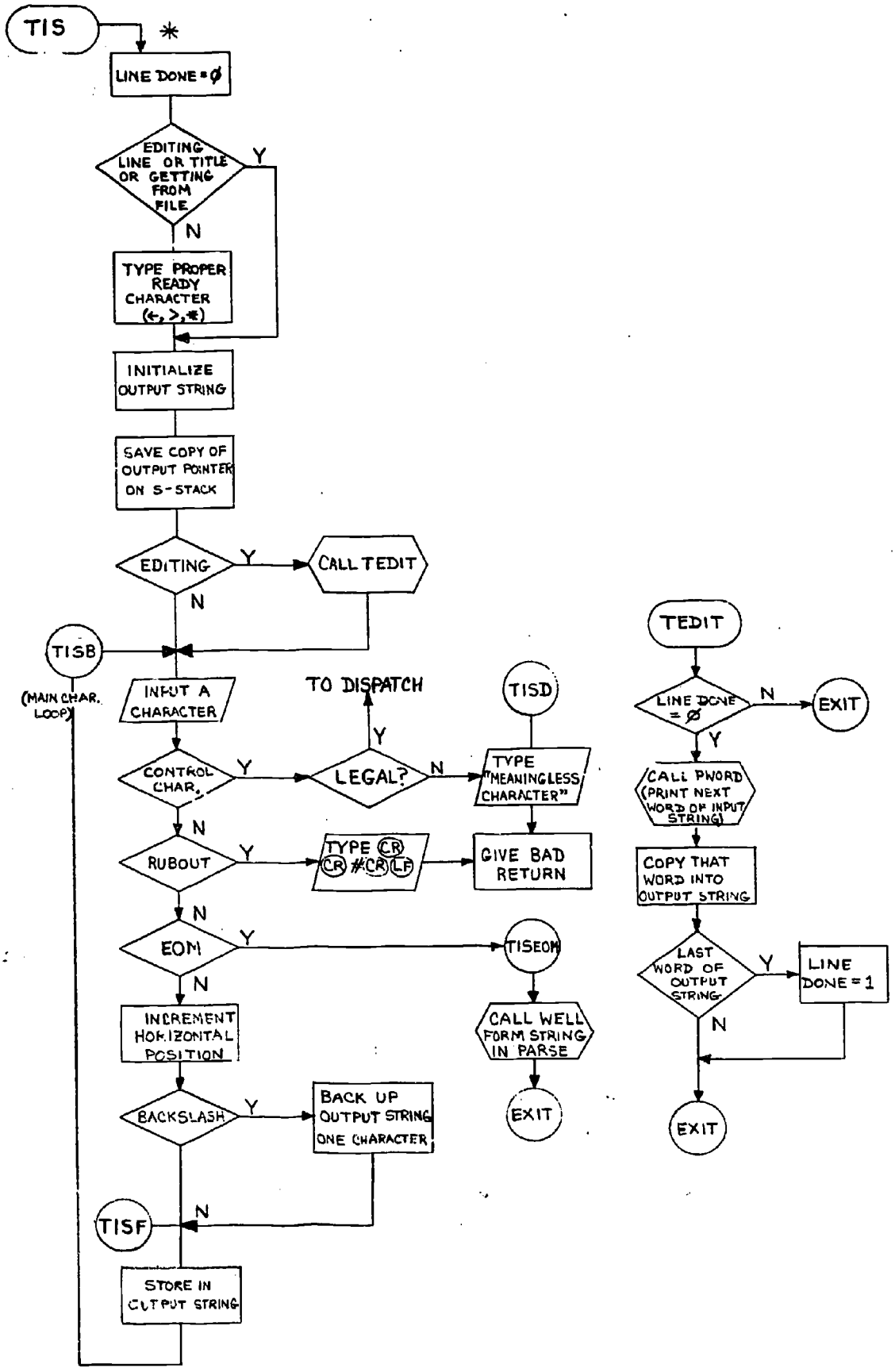
PARSE - 3


LIT
COM
LAST CHAR. IS TERMINATOR
CALL WEFS
ENDLIT
LIST ELEMENT IS PTR TO THE STRING
SAVIT
SAVIT
STORE ELEMENT IN PARSED LIST
GNS
VAR
/ IS THE TERMINATOR
CALL WEFS
IS THERE AN ENTRY IN NAME TABLE WITH THIS AS THE NAME
YES
NO
ADD AN ENTRY TO NAME TABLE WITH THIS STRING AS THE NAME
LIST ELEMENT IS PTR TO THE ENTRY
SAVIT

PARSE - 4

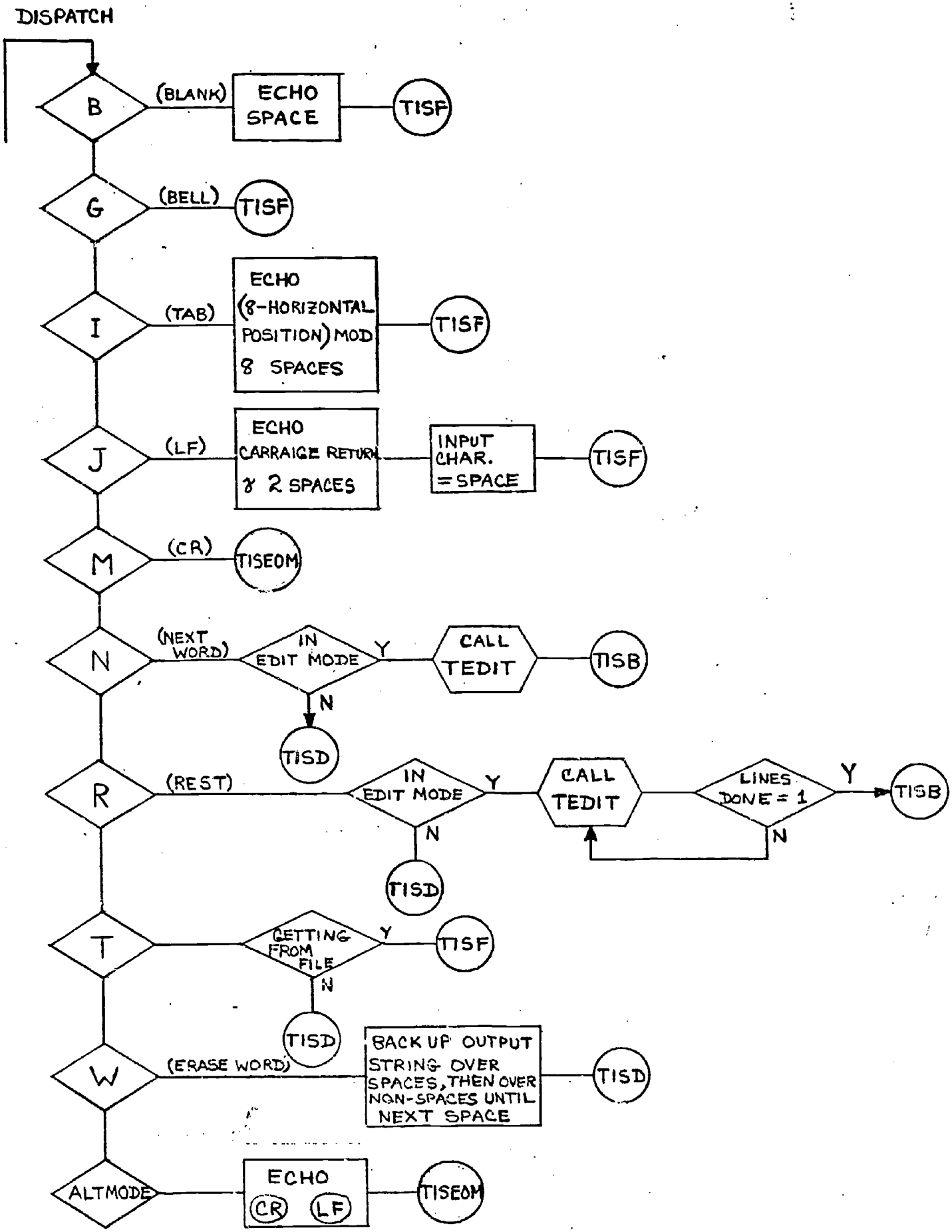

LP
LIST ELEMENT IS BUILT-IN (
SAVIT
RP
LIST ELEMENT IS BUILT-IN)
SAVIT
EOA
DEPTH OF ABBREVIATION = Ø
NO
DECREMENT DEPTH OF ABBREVIATION
POP A PTR TO INPUT STRING
GNS
YES
WAS FIRST LIST ELEMENT AN UNSIGNED NUMBER
NO
EXIT
YES
IS TOPROD = Ø
YES
ERROR
NO
STORE LINE NUMBER AND PTR TO PARSED CODE IN PROCEDURE DIRECTORY
RETURN TO INPUT


WEFS
(WELL FORMED STRING ENTERED WITH A SPECIFIED TERMINATOR)
FETCH NEXT CHAR.
IS IT SPACE
YES
NO
WEFS
IS IT THE SPECIFIED TERMINATOR
YES
FINISH OUTPUT STRING
EXIT
NO
WEFD
IS IT EOM
YES
IS THE TERMINATOR ;
YES
NO
ERROR
NO
STORE CHAR. IN OUTPUT STRING
FETCH NEXT CHARACTER
IS IT SPACE
YES
NO
IS IT THE TERMINATOR
YES
WEFB
NO
IS IT EOM
YES
WEFD
NO
STORE A SPACE IN OUTPUT STRING

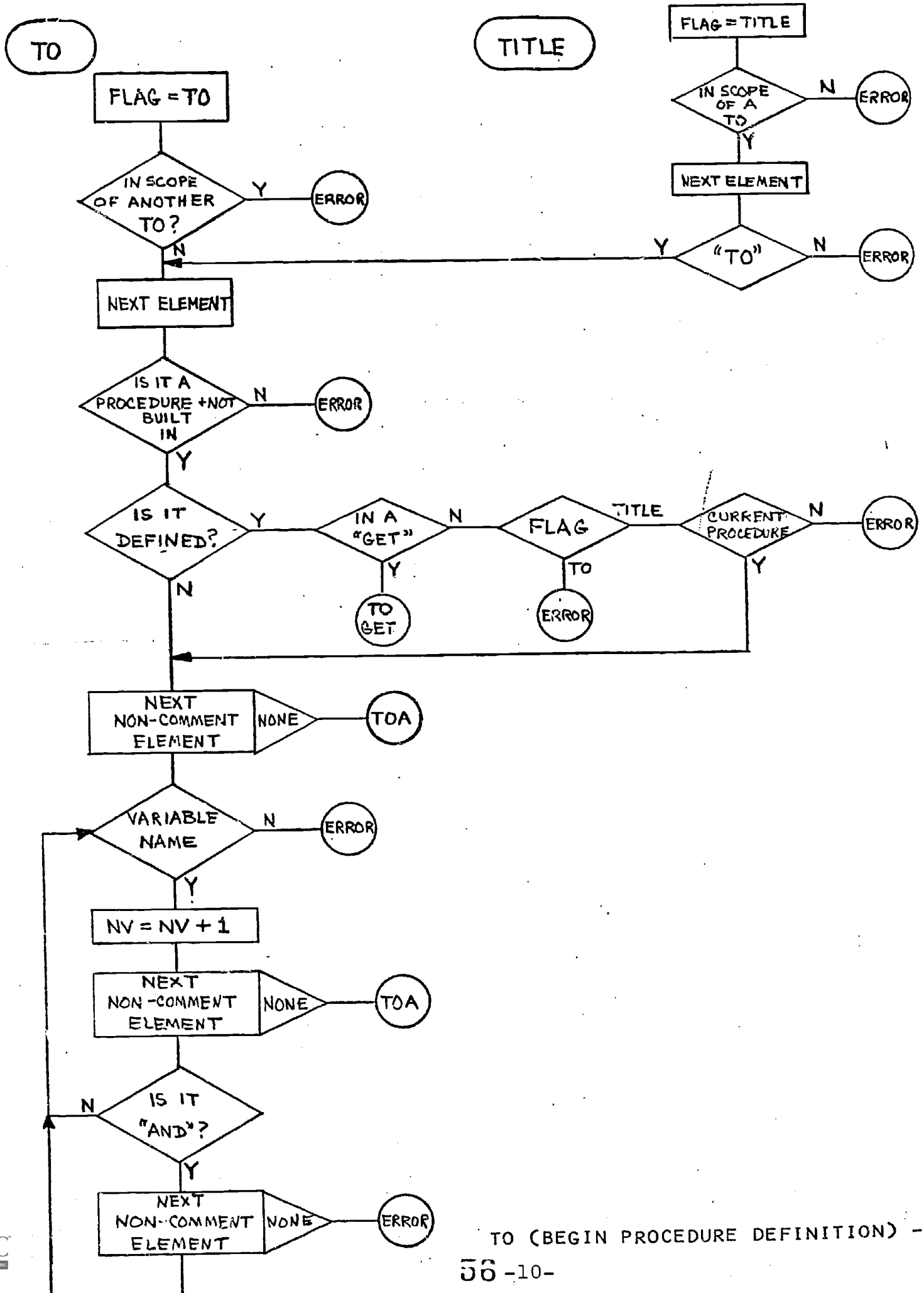


* IN EDIT MODE, TIS IS ENTERED WITH AN INPUT STRING- WHICH IS THE LINE TO BE EDITED.

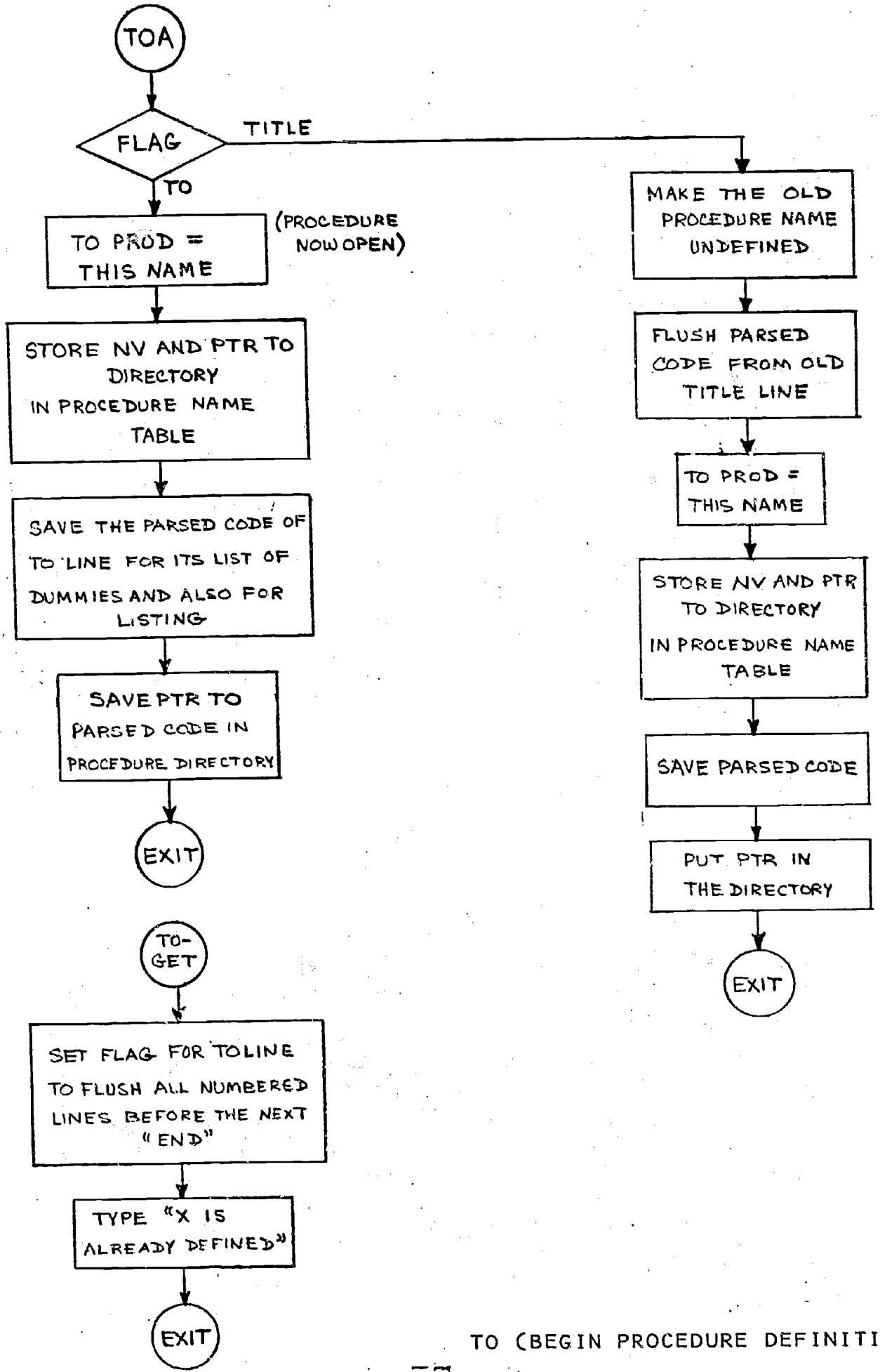
TYPE IN STRING (TIS)



TYPE IN STRING (TIS) -2



TO (BEGIN PROCEDURE DEFINITION) - 1



TO (BEGIN PROCEDURE DEFINITION) - 2